REMARKS

Applicant wishes to thank the Examiner for the attention accorded to the instant application, and respectfully requests reconsideration of the application as amended.

Formal Matters

Claims 76-98 are the claims currently pending in the Application. Claims 76 and 98 are amended herein to more clearly recite the invention.

Response to Amendment

The Examiner objects to the amendment filed on August 4, 2006 under 35 U.S.C. § 132(a) because it allegedly introduces new matter into the disclosure. The Examiner contends that the step of "relocating all instructions and all variables ..." is not supported by the original disclosure and request clarification. Applicant amends claims 76 and 98 and clarifies support for this step as follows.

The aspect of relocating variables, that is, "relocating all instructions and all variables within said executable file affected by the insertion", as recited in claims 76 and 98, is disclosed in paragraph [0141] of the published application. This paragraph discusses insertion of data between existing subroutines and variables, respectively. In particular, it addresses the requirement of relocation by stating that variables that are "located at higher addresses than the selected one are then moved towards higher addresses by the number of bytes to be inserted" (paragraph [0141], lines 7-10). The variables that are "located at higher addresses" are exactly those variables that are "affected by the insertion" according to the amended claims; these variables are "moved towards higher addresses" or relocated. After having discussed insertion between

subroutines and variables, the next paragraph [0142] introduces the idea of insertion into the code of subroutines.

The aspect of relocating instructions is disclosed in paragraph [0144] of the published application. The paragraph notes that the subroutine into which code has been inserted "grows in size". Hence, if "a3" denotes the address of the subroutine into which code is inserted and "N" denotes the number of bytes that have been inserted, all subroutines located at "higher addresses than a3 are then moved towards higher addresses by N bytes". The code at higher addresses than "a3" comprises exactly those instructions that are "affected by the insertion" according to the amended claims.

The recitation of "adjusting all of the one or more instructions and all of the one or more variables within said executable file that relate to the relocated instructions or variables" is also disclosed in paragraphs [0141] and [0144]. Insertion between variables is covered by paragraph [0141]. In particular this paragraph says that finally "all references concerned are readjusted" (line 12). Regarding insertion into subroutines, paragraph [0144] notes that finally "all references to the offset subroutines are adjusted" (line 13). Thus, the specification discloses the step of "relocating all instructions and all variables within said executable file affected by the insertion, and adjusting all references to the relocated instructions and/or variables to reflect the relocating of the instructions and variables" and no new matter has been added. Applicant respectfully requests that this objection be withdrawn.

Response To Rejections Under 35 U.S.C.§ 102

The Examiner rejected claims 76-98 under 35 U.S.C. § 102(e) as being anticipated by Biddle et al., U.S. Patent Publication No. 2002/0107809 A1 (hereinafter

"Biddle"). This rejection should be withdrawn based on the comments and remarks herein.

Among the problems recognized and solved by Applicant's claimed invention is the need for an automated technique for performing an extension of an executable file to extend its functionality. One approach to this problem has been to change the source code underlying the executable and then compile the changed source code. The only known alternative approach of adding functionality to an executable file after compilation and without modifying the source code of the executable is the technique known as "wrapping", which adds functionality that executes before the original functionality of the executable file. In contrast to the known technique of changing the source code, the present invention operates exclusively on the executable file and, unlike wrapping, functionality is added *within the existing code* and not only before the original functionality.

Biddle discloses the technique known in the art as "wrapping". Where the term "code injection" is used in context with wrapping in Biddle, this will be understood by those skilled in the art as adding code before the existing executable file. Wrapping can be explained as follows. To successfully run an executable, an operating system has to know where in the code of the executable execution is to start, i.e., to which instruction inside the executable the operating system should jump, after the executable has been loaded into memory. This entry instruction is identified by meta-data inside the executable. This meta-data is commonly known as the "program entry point".

For example, suppose that we have an original executable that contains a code segment with a length of 1000 bytes, which is located in the address range 0 through 999.

Suppose that the instruction with which to start program execution is located at address 102. This gives us a program entry point of 102.

The idea underlying wrapping is the following. The 1000 bytes of the code segment are considered as an opaque sequence of bytes instead of individual instructions. Wrapping does not care what these bytes represent because wrapping only adds new code that is executed *before* the original code and this can be accomplished without understanding and altering the 1000 bytes of the code segment. Wrapping simply appends new code, or "start-up code" as Biddle calls it, to the original code segment.

Further, suppose that the start-up code consists of 200 bytes. These 200 bytes would be appended to the original executable and would thus be located in the address range 1000 through 1199. The new code segment would thus consist of the 1200 bytes in the address range 0 through 1199. However, the appended start-up code must be executed before executing the original code. Hence, in wrapping, the final step is to rewrite the original program entry point in the meta-data of the executable file to point to the entry instruction of the start-up code. In this way, the start-up code will be invoked when the wrapped executable is run. Once the start-up code has finished execution, it will jump to the original program entry point of 102, which will start execution of the original code.

To sum up, wrapping is a method of sequential execution of (1) start-up code newly added to an executable, and (2) the original code of the executable. This is accomplished without analyzing or modifying the original code segment. The original code segment is handled as an opaque sequence of bytes.

The problem with wrapping as described above is that an attacker could simply change the program entry point of the wrapped executable back to its original value of 102. If the modified executable were executed, it would not run the start-up code but the original code, as 102 identifies the instruction with which to start execution of the original code. As the start-up code often implements license enforcement, changing the program entry point back to its original value would eliminate license enforcement, yielding an unprotected executable.

Consequently, there has been a lot of innovation around the question of how to *ensure* that the start-up code is actually run. Typically, the original code segment, i.e. the address range 0 through 999, is encrypted. The start-up code in the above example would then contain a decryption routine that decrypts the original code segment before jumping to the original program entry point of 102. As only the decrypted original code can be executed and because only the start-up code contains the decryption routine, the start-up code must run before the original code can be executed. This is described in Biddle (paragraph [0082], lines 18-23) as an exemplary embodiment, as discussed in more detail below.

Encryption only protects the privacy of information. To protect the original code against tampering, the start-up code typically calculates checksums over regions of the original code and compares the calculated checksums to the checksums expected for unmodified code. If the checksums match, the code is considered untouched by an attacker. Checksums are typically calculated using cryptographic hash functions.

Note that encryption, decryption, and checksum calculations are possible despite the fact that wrapping considers the code segment to be an opaque sequence of bytes.

These security primitives do not require analysis of the code segment. It is not necessary to know the structure of the data, or to identify the instructions and/or variables represented by the encrypted, decrypted, or checksummed sequence of bytes.

The Examiner cites paragraph [0082] of Biddle as disclosing a technique that reaches beyond wrapping. Applicant respectfully disagrees. As discussed above, Biddle describes a typical wrapping scenario with start-up code that decrypts the original code using itself as the decryption key and, at the same time, checksumming the original code to guard against tampering. The start-up code decrypts the original code segment and calculates checksums based on cryptographic hashes which are calculated in the process. Biddle also describes another application of the wrapping notion of considering code as a sequence of bytes by suggesting that the start-up code be used as the key for decryption, since a cryptographic key used for decryption is a sequence of bytes. As discussed above, the idea behind using code as a decryption key is that illegitimately modifying the code to circumvent a licensing scheme will change the key for decryption. However, if anything but the correct key is used for decryption, decryption will fail, i.e. the original code segment will not be correctly decrypted and thus will not be executable.

Wrapping, however, is completely different from the present invention. As discussed above, wrapping assumes the original code segment to be a sequence of bytes which are not examined or analyzed. In contrast, the present invention analyzes the original code segment to identify its contents and, based on the gained insight, inserts instructions into it. Accordingly, claim 76 recites "identifying one or more instructions and/or one or more variables within the executable file" and "inserting data and/or one or more instructions within said executable file".

The above also applies to variables. Wrapping the data segment typically means encrypting it and having the start-up code decrypt it. Again, a wrapper does not try to identify individual variables in the data segment. It simply considers the data segment to be a sequence of bytes that can be subjected to encryption, decryption, and checksumming. In contrast, the present invention identifies instructions and variables, including the boundaries between variables for the purpose of insertion, which reaches beyond what wrapping is able to accomplish.

Biddle, paragraphs [0055-0056, 0076-0077, 0091-0104], discloses a modern system for license enforcement that combines the strengths of an API-based plug-in solution that is manually integrated by the software vendor into the source code of the application to be protected with the standard protective technology of wrapping. Biddle does not disclose or suggest identifying either instructions or variables, or inserting instructions and/or variables between other instructions and variables as recited in the independent claims.

In the Examiner's opinion, paragraph [0074] of Biddle discloses or suggests the present invention (Office Action, page 4, line 9 to page 5, line 2). Applicant respectfully disagrees. Paragraph [0073] of Biddle provides the context for the embodiment presented in paragraph [0074]. This paragraph introduces what is known in the art as a *plug-in* architecture. Plug-ins are software modules that extend the functionality of a given application. An application that supports plug-ins offers a well-defined interface (API - Application Programming Interface) that enables the application to exchange data with its plug-ins, that enables the plug-ins to invoke functions inside the application, and that enables the application to invoke functions inside the plug-ins. Typically, the

application loads available plug-ins via a dynamic linking mechanism such as DLLs (Dynamic-Link Libraries) on the Microsoft Windows platform. So, exchanging the implementation of a plug-in for a different implementation is typically as easy as replacing a single file, e.g. a single DLL.

Paragraph [0073] of Biddle discusses an API that the software vendor programs against in order to implement license enforcement. It is important to note that this implementation is done manually by the vendor in his or her source code, in contrast to the invention at hand, which operates exclusively on executable files. The implementation consists of adding invocations of licensing API functions to the source code. To improve resistance against attacks, Biddle suggests that the invocations of the API functions be "placed by the vendor at strategic locations throughout the code" (paragraph [0074]), "code" meaning "source code".

The novelty of this aspect of Biddle lies in the fact that, just like a plug-in, the code that implements license enforcement can be easily exchanged by authorized parties So, by programming in accordance with the API, the vendor software becomes an application that supports licensing plug-ins. The advantage of the plug-in paradigm is that during development, a dummy plug-in ("the toolkit's test version of the licensing calls and the licensing manager", paragraph [0074]) that unconditionally allows the vendor software to run without validating any licensing conditions can be used for testing. Once development is complete and the vendor software is ready to be distributed, the dummy plug-in can be substituted by a production plug-in ("the original version of the distributor software", paragraph [0074]) that does enforce licensing conditions.

Biddle suggests that it does not have to be the software vendor that replaces the dummy plug-in. It could as well be the distributor, so that Biddle "reduces the vendor involvement in security violation issues and transfers the responsibility to the distributor" (paragraph [0074]). However, if the distributor is responsible for license enforcement, it makes sense to also leave wrapping to him or her and, thus, to integrate wrapping and substitution of the dummy plug-in, so that the API calls that the vendor has added to his or her source code "are substituted during the wrapping process with a final version of the licensing API code provided by the distributor" (paragraph [0073]).

Thus, as an alternative to wrapping, Biddle discloses a "plug-in architecture" for license enforcement code. It is based on an API that has to be integrated by the software vendor into the source code of his or her application to be protected. This does not relate to the invention at hand, which exclusively processes executable files and does not operate on source code.

It has been held by the courts that "Anticipation requires the presence in a single prior art reference disclosure of each and every element of the claimed invention, arranged as in the claim." *Lindemann Maschinenfabrik GMBH v. American Hoist and Derrick Company et al.*, 730 F.2d 1452, 221 USPQ 481 (Fed. Cir. 1984). As illustrated above, Biddle does not disclose each and every feature of the invention as recited in independent claims 76 and 98. Claims 77-97 depend from independent claim 76, thus incorporating novel and nonobvious features of the base claims. Accordingly, claims 77-97 are patentably distinguishable over the prior art for at least the reasons that independent claim 76 is patentably distinguishable over the prior art. Therefore, this rejection should now be withdrawn.

Conclusion

For at least the reasons set forth in the foregoing discussion, Applicant believes that the Application is now allowable, and respectfully requests that the Examiner reconsider the rejection and allow the Application. Should the Examiner have any questions regarding this Amendment, or regarding the Application generally, the Examiner is invited to telephone the undersigned attorney.

Respectfully submitted,

Katherine R. Vieyra
Registration No. 47,155

SCULLY, SCOTT, MURPHY & PRESSER, P.C.
400 Garden City Plaza, Suite 300
Garden City, New York 11530
(516) 742-4343

KRV/vh